8-14-1986

Combined query language (CQL): an interactive query language with procedural features

Ilia Levi

Rochester Institute of Technology
School of Computer Science and Technology

Combined Query Language (CQL):

An interactive query language with procedural features

by
Ilia Levi

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Jeffrey Lasky
Jeffrey A. Lasky — 8-14-86 Date

Lawrence A. Coon
Lawrence A. Coon — 8/14/86 Date

Guy Johnson
Guy Johnson — 8/14/86 Date

Abstract:

Interactive query languages provide easy to use interfaces to database systems. However, they lack the power of the host languages to perform manipulations on data.

This thesis presents an interactive query language Combined Query Language (CQL) that has some features of procedural languages. CQL allows users to interactively manipulate databases and perform calculations on data.

CQL is a self contained interactive query language that combines the computational power of a host language (like FORTRAN) with the convenience and flexibility of the common interactive query languages.

Users of CQL would not have to use a host language for most of the database applications.

CQL uses the low level data base management routines developed for another thesis.'

' Robert Fabbio's masters thesis.

Acknowledgement:

I wish to thank the members of my thesis committee:

Jeffrey A. Lasky,

Lawrence A. Coon,

Guy Johnson

for all the help with this thesis and also my wife Alla for putting up with me while I was working on it and proofreading this document.

Table of Contents

List of Illustrations

CHAPTER 1

Project Description

1.1 Types of Query Languages and Their Limitations

There are many Database Management Systems in use at this time. Most of them provide specialized query languages to access the data. The two types of query languages are:

- Interactive Query Languages -- these languages typically do not require data declarations, have very high level and specialized functionality, and eliminate or minimize need for user to express procedural requirements.

  The Interactive Query Languages are primarily designed for ad hoc queries. Typically, they provide minimal computational capabilities.

- Embedded Query Languages -- these languages are embedded within the programs written in the host languages. The syntax of the query language is either compatible with the host language or a preprocessor is used to convert the EQL statements into the host statements that invoke DBMS functions. The host language is a language that can accommodate the query language statements. Normally, general purpose languages are used, such as PL/1, FORTRAN or C. The programs are compiled and run. Any changes made to the queries require changes to be made to the programs, which then have to be recompiled. Typically, the syntax of the interactive query languages very closely resembles the syntax of the host languages. The Embedded Query Languages can utilize the full functionality of the host language, but are costly for ad hoc queries and naturally force the user to be a programmer conversant in the host language.

With the conventional systems, in order to utilize the full capabilities of the DBMS the user has to learn both the Embedded and Interactive Query Languages, a host language, and the interface between the Embedded Query Language and the host language.

1.2 Combined Query Language (CQL)

CQL is an Interactive Query Language that combines the computational power of an embedded query language with the convenience and flexibility of an interactive query language. It is an interpretive language that allows dynamic declaration of local variables for intermediate computations during a query session.

As will be seen, CQL is unique in respect to providing, without looping constructs, both set and record-at-a-time processing. This is accomplished through the INVOKE option of the SELECT statement. It invokes a CQL subroutine for each selected tuple.

For example, using CQL, a conditional record retrieval command is issued. As a part of the command, the user can specify a CQL subroutine to receive control for each selected record. In this example the user needs to determine the two highest paid programmers on the project 'x'.

The following commands are issued interactively:

```
SET highest = 0;
SET nexthigh= 0;
SELECT employee.project = "x" and
       employee.title = "programmer",
  INVOKE max2, ATEND max2end;
```

MAX2 is a CQL subroutine that is invoked for each selected record. It looks like this:

```
if employee.salary > highest
{
     set nexthigh=highest;
     set highest=employee.salary;
};
else if employee.salary > nexthigh
     set nexthigh=employee.salary;
```

MAX2END is a CQL subroutine that is invoked after all records were selected. It consists of the following statement:

```
print highest nexthigh;
```

CQL calls low level DBMS routines to access the databases. The low level routines were written as part of a previous RIT thesis. |12|.

CQL has statements commonly found in the query languages, such as SELECT to choose a subset of tuples that meet certain criteria, and OPEN to establish a link to the databases.

In addition, it has statements that are found in procedural languages. These statements perform computations on the data in the database and on local variables (created by CQL the first time they are encountered in the program) and control the order of execution of the statements. Examples of such statements are assignment and WHILE statements.

1.3 Basic Constructs of CQL

In order to define CQL, we need to introduce several basic elements of the language that will be used later.

1.3.1 variable

A variable is either a local variable or a database variable. Local variables are created when CQL first encounters them during execution. Once created, they remain active until the end of the session.

Database variables identify the fields in the database. They contain values of the current tuple.

Both types of variables can attain integer or character string values. Integer variables are four byte variables containing signed integers, while character variables contain character strings terminated by a null character. The type of database variable is fixed by the type of the corresponding field in the database, while the type of local variable is dynamic and can be integer or character, depending upon what value is assigned to the variable.

1.3.2 expression

Expressions in CQL are infix expressions that evaluate to an integer, logical or character string value. The logical val-

ues are actually integers where any non-zero value signifies TRUE, and zero means FALSE. The allowable operators are:

Binary arithmetic: '*', '/', '+', '-'. The '*' and '/' have precedence over '+' and '-'.

Binary logical: 'AND', 'OR'. Both have equal precedences.

Unary NOT: '!'

Binary Comparison: '=', '!=', '>', '<', '>=', '<='. All have equal precedences.

The '!' has the highest precedence, followed by the arithmetic operators, followed by the comparison operators. The logical operators have the lowest precedence. The order of evaluation can be changed by using parentheses. The operands can be either variables (local or database) or constants. The only operations allowed on the string variables are comparisons.

1.4 CQL Statements

In addition to the BNF notation of CQL presented in appendix A, this section includes the description of CQL using a different, less formal notation. The notation has the following rules:

Uppercase words and special characters, except for the vertical bar '|', are required syntax. Anything surrounded by the '|' is optional. If such a surrounded construct is followed by '...', then it can be repeated as many times as desired. Lowercase words specify that a basic construct of CQL (see section A.0) is required.

1.4.1 CASE statement

```
CASE expression1
{
    (subexpression |,subexpression|...):statement
   |(subexpression |,subexpression|...):statement|...
   |OTHER: statement|
};
```

The CASE statement first evaluates expression1. Then, one by one, it evaluates the subexpressions until it finds one whose value matches the value of expression1. If such a subexpression is found, the statement associated with the subexpression is executed. Control passes to the next CQL statement. If no such subexpression is located, the statement in the OTHER clause, if specified, is executed.

1.4.2 CLOSE statement

```
CLOSE dbname |,dbname|...;
```

The CLOSE statement makes calls to the underlying file structure routines to close the files associated with the databases and to free the control blocks allocated when the databases were opened.

1.4.3 EXIT statement

```
EXIT;
```

EXIT returns control to the OS shell. If there are any files open at this time, they are closed.

1.4.4 FOR statement

```
FOR variable = expression1 TO expression2
                          |STEP expression3|
    statement
```

This statement evaluates expression1 and assigns the result to the variable. Then it evaluates other expressions and stores the results in the temporary variables. The default value for expression3 is 1. The value of the variable is then compared to the value of the saved result of the expression2. If it is greater, the FOR is finished. Otherwise the statement is executed; the sum of the variable and the saved value of expression3 is assigned to the variable; then the comparison is (made again, ...) until the value of the variable exceeds the value of expression2. Note that the check is done before the statement is executed, so that the statement inside the loop may never be executed.

1.4.5 IF statement

```
IF expression
   statement1
|ELSE statement2|
```

If the result of the expression if TRUE, statement1 is executed, if it is FALSE, statement2 is executed if it exists.

1.4.6 INVOKE statement

```
INVOKE identifier;
```

The INVOKE statement executes a CQL subroutine from the file whose name is formed by the identifier followed by a period followed by the letter 'm'. The subroutine can be composed of any valid CQL statements.

1.4.7 OPEN statement

```
OPEN dbname |,dbname|...;
```

The OPEN statement makes calls to the underlying file structure routines to open the files associated with the databases and to allocate the control blocks that describe the databases.

1.4.8 PRINT statement

```
PRINT variable |,variable |...;
```

PRINT is a very rudimentary print statement. It displays the values of the variables, one per line.

1.4.9 REPEAT statement

```
REPEAT
   statement
UNTIL expression;
```

The statement inside the REPEAT loop is executed until the value of the expression becomes TRUE. Note, that the statement is executed first, and then the expression is evaluated, so the statement will be executed at least once.

1.4.10 SELECT statement

```
SELECT special-expression
   |,PRINT variable |,variable|... |
   |,INVOKE filename|
   |,ATEND filename|
   |,SAVE filename|;
```

Where the special-expression is of the form

```
 database variable = constant | AND expression |
```

The SELECT statement retrieves tuples that satisfy the special-expression from the database. If no options are specified (PRINT, INVOKE or SAVE) the PRINT is assumed.

1.4.10.1 PRINT option

For each tuple, the PRINT option causes values of the specified variables to be displayed, one per line.

1.4.10.2 INVOKE option

For each tuple that is retrieved, the INVOKE command is issued. The subroutine can then process the tuple.

1.4.10.3 ATEND option

After all tuples are processed, the INVOKE command is issued.

1.4.10.4 SAVE option

This option will perform the same function as the PRINT option, except it will print into the file identified by the filename.

1.4.11 SET statement

```
SET variable = expression;
```

The value of the expression is assigned to the variable.

1.4.12 SYS statement

```
SYS any UNIX command;
```

A UNIX command is executed. One of the uses for this statement maybe

```
SYS CSH;
```

Which will give control to the shell, and let the user to issue any number of UNIX commands. Control-D returns control to CQL. One can invoke an editor and create a CQL subroutine.

1.4.13 WHILE statement

```
WHILE expression
   statement
```

The statement inside the WHILE loop is executed only as long as the value of the expression is TRUE. The expression is

evaluated before the statement is executed, so the statement may never be executed.

1.5 Components of CQL

The CQL consists of 5 parts: the parser, lexical analyzer, syntax tables, executor and the low level DBMS routines.

1.5.1 Parser

The parser takes the statements and translates them into tokenized objects. The parser is table driven, so it is relatively easy to add new, or modify existing, CQL statements.

1.5.2 Executor

The executor takes the tokenized objects produced by the parser and executes them. Any new CQL statements would require additional code in the executor to handle the statement.

To access the databases, the executor calls the subroutines provided by the underlying "Relational-like Database File

Structure" developed by Robert Fabbio in his masters thesis. |12|

1.6 Limitations

The CQL is not designed to do the following:

- Update the databases. This is because the underlying file service routines do not provide this capability.

- Connect tables. We want to show in this thesis how a combination of the features of the interactive and embedded query languages can be accomplished. Connection between tables does not contribute anything to the problem and goes beyond the scope of this thesis.

- Selection is restricted to the keyed field, which is optionally further restricted by the expression. No alternative for the key value is allowed. Also, no nested SELECTs are allowed. These limitations are here for the same reason as in the previous item.

CHAPTER 2

Evolution of Query Languages

Commercial database management systems were not offered until the mid to late 1960s. The reasons for such a relatively late arrival were:

- Unattractive price/performance ratios for mainframe class CPUs
- High memory (both main and disk) costs
- Immature database theory

These made the general DBMSs slow, expensive and relatively inflexible. It was cheaper and more practical to write specific programs for individual applications. As the price/performance ratio of systems improved, and the benefits of general purpose database management systems became more apparent (reduction of redundancy, avoidance of inconsistency, enforcement of standards, improved security and integrity, centralized control of data), such systems were introduced.

2.1 Types of Query Languages

The first database systems used the hierarchical (e.g. IMS) and network (e.g. CODASYL) data models. Users of these systems had to understand the complex interactions between records. In 1970, Codd published a theoretical paper that created much interest. It introduced the concept of the relational model |1|. The relational model provides high degree of flexibility and data independence. Since the relational DBMS does force the user to specify the relationships between the records during the definition of the database, queries can be very flexible and easy for the users to understand.

To provide a user friendly interface to the DBMS, interactive query languages were developed. These languages are high level computer languages which are oriented toward the manipulation of data stored in the databases |8|.

There are two types of query languages:

1. Embedded Query Languages: these are extensions to the host languages through subroutine calls or preprocess-

ors. These extensions provide a facility to retrieve, and usually update, data in the databases.

2. Interactive Query Languages: these are stand-alone languages that in addition to the statements for retrieval and update of data, also provide for displaying of the results, usually through a report generator.

The query languages can manipulate data in two ways:

o Record-at-a-time: conventional file systems and many early database systems use this type of retrieval. It looks like a conventional data retrieval in procedural languages. The records are retrieved one at a time, processed and the next record that matches the selection criteria is fetched. An example of such languages is TYMSHARE's MAGNUM. MAGNUM uses the relational model and provides access to the data in interactive and batch modes. It provides full computational and report generation facilities |3|.

Record-at-a-time languages need to have looping constructs to process queries with multiple records. This makes them difficult to use by non-programmers.

- Set-at-a-time: the introduction of the relational model led to set-oriented data retrieval. These languages retrieve all records that satisfy the query as a part of request and process the created set. Many problems are better expressed in terms of sets rather than in terms of individual records. The query languages that manipulate sets are usually higher level than the record-at-a-time languages. They tend to be non-procedural and are better suited for the non-programmers and ad-hoc queries. In addition, the advantages include:

  - no need for looping constructs

  - simplicity -- most queries are easier to solve by a set oriented language.

2.2 Classification of Query Languages

The query languages were developed from two different directions:

1. The need for simple end user interface
2. From the directions of research in programming languages and databases.

Figure 1 on page 32 depicts the development trends of query languages and shows query language usability vs. QL functional capability |13|.

The first approach led to the development of the query languages that constructed queries (in the order of increased usability and functional capability) by the following methods:

1. Function keys. This method is particularly good for creating a limited, yet effective system for unsophisticated users.

2. Line-by-line prompting. This is simple dialog between the user and the system. By answering questions, the user narrows down his request until the system has sufficient information to construct a query. The user maybe prompted for the objects of interest, values and data relationships between the objects.

3. Menus. They are a more sophisticated method, similar to the line-by-line prompting. The user chooses the options provided by the menus, fills in the values and field names. Some menus, depending on the options chosen, may lead to other menus.

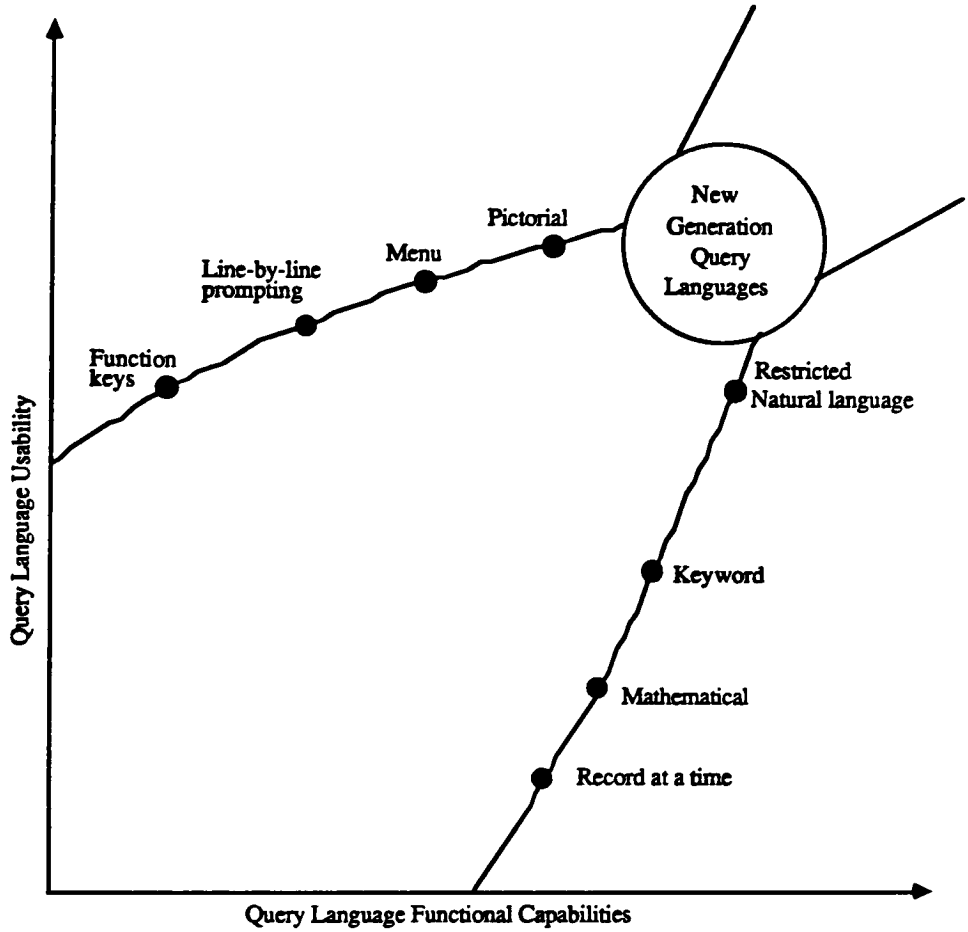


Figure 1. Query Language Development Trends. (From |13|)

4. Graphic and pictorial query languages. The user manipulates the virtual symbols to request data from the database. The objects and relationships them are represented by geometric symbols. Such systems require advanced graphical hardware. One good use for such systems is in the area of chemical databases, where the chemical compounds can be readily represented by the chemical symbols.

The first three methods are most useful for the custom-made languages for specific applications, whereas the fourth method is suitable for the systems that need to manipulate the data that can be readily represented symbolically.

The second approach provided these types of query languages:

1. Record-at-a-time. The most simple of these query languages, yet the most difficult to use for the non-expert user.

2. Mathematical. These query languages use the mathematical notation for short and powerful expressions. Two of such systems are ALPHA and PASCAL/R. These languages are particularly useful as intermediate languages for

the very high level user interfaces, since many users of the database systems are not well versed in mathematical formalism.

3. Linear keyword. Most of the todays query languages fall into this category. These languages are similar to the conventional languages like C, but are more English-like and are more user friendly. These languages typically do not include the procedural constructs of the general purpose programming languages. Examples of such QLs are SQL (IBM's DB2) and QUEL (INGRES).

4. Restricted natural languages. These languages use a subset of natural languages, such as English, to communicate the user's requests to the system. Some systems can ask questions resolve any ambiguities that the user introduces into his query, since natural languages are notoriously ambiguous. Some restricted natural language query languages are front ends to the conventional linear keyword languages, such as SQL. The user may get a chance to review the constructed query in such a languages before its execution.

Both of the approaches just discussed are classified as the "previous generation languages".

The "new generation languages" attempt to provide an interface that allows the user to use his instincts and senses.

The "intelligent" systems use artificial intelligence techniques to derive the user's needs. These systems try to bring the interaction between the user and the computer close to the level of interaction between the people.

The "direct manipulation systems" provide the user with the rapid reversible actions, visible objects of interest and replacement of the keyword language by direct manipulation of objects |13|.

The next sections examine some existing interactive query languages.

2.3 System 2000

System 2000 from MRI Systems Corp. is a hierarchical DBMS that provides an interactive user language interface. It supplies report generation and some standard statistical

functions on sets, such as COUNT, SUM, AVERAGE, MAXimum. An example of the statement of the System 2000 QL is:

```
PRINT state name WHERE state name HAS
    city name EQ rochester
```

The result is New York and Minessota. In this database, the parent database state has children cities. The HAS operator is necessary to specify the parent/child relationship. While this QL is easier to use than IMS, where one has to specify to the system how to retrieve records, the user still has to understand the hierarchy of data, and be careful when generating the query to specify the parent/child relationship correctly |7|. Also, some questions would be difficult to ask, because the hierarchical model is not very flexible.

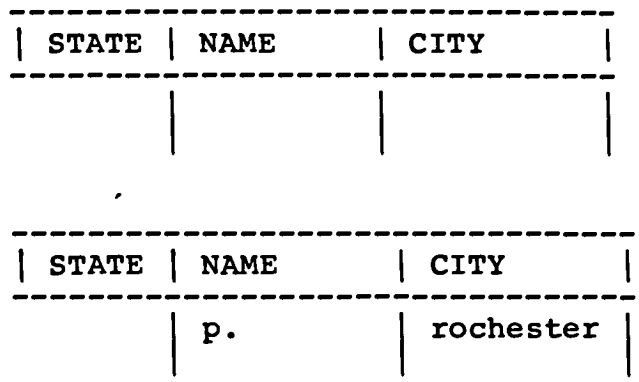
2.4 Query-by-Example

One of the first commercial interactive query products for the relational DBMS was Query-by-Example (QBE). It was first introduced by M. Zloof at the National Computer Conference in 1975, and as a more refined system in |2|. IBM introduced QBE as a commercial product in 1978. QBE is a relational system that was designed specifically for use

with a video display terminal. QBE provides the user with a powerful tool to manipulate the relational databases, while letting the user think in terms of tables, rows and columns. The user fills out portions of the skeleton tables on the screen by moving the cursor to the desired fields and entering data, and QBE fills in the rest. QBE is easy to learn and use, and it is very well suited for non-programmers. QBE is a 2-dimensional language. An example of the query from the previous section is shown in Figure 2 on page 38(user entries are in lower case, system responses are in uppercase):

2.5 DB2

DB2 is a commercial implementation derived from System R |3|. It is another relational system with a powerful interactive query language that was developed by IBM between 1974 and 1979 at San Jose research laboratories. The interactive language that DB2 uses is SQL (Standard English Query Language). SQL is based on the earlier language SQUARE that has much more mathematical syntax, but is essentially the same, except SQL's syntax was modified to be more English-like. SQL is designed for both programmers and non-programmers. In addition to the interactive query language, SQL has an em-



The p. requests QBE to print the state name.

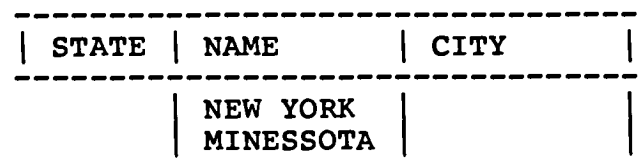


Figure 2. Example of QBE session

bedded form |3|. SQL has been adopted by many other systems; for example, Unify, Oracle and IBM's SQL/DS.

A previous example in SQL looks like this:

```
SELECT name
FROM   state
WHERE  city = 'rochester'
```

2.5.1 Standard SQL

The growing use of SQL prompted the formation of an ANSI Database Committee (technical committee X3H2) to specify a SQL standard. According to my conversation with the chairman of the committee, Donald Deutsch |11|, the result was the least common denominator of all implementations of SQL, since the companies that adopted SQL as their language could not agree which features to make standard. They were reluctant to add a feature to SQL that their implementation did not possess, but their competitor's version did have. The least common denominator is the IBM's implementation of DB2.

2.5.2 Comparison of SQL and QBE

QBE and SQL provide similar power when used interactively. This poses an interesting question of which language is more effective in manipulating the database. SQL is a linear language (commands are entered in a linear fashion, one at a time), whereas QBE is 2-dimensional(columns on the screen are filled in, and QBE completes the rest).

Some research was done recently in comparing the ease of use of these two languages by non-experts. The results were in-

conclusive. SQL was judged easier to use for arithmetic calculations, while QBE was simpler to use for joins and selections |4|.

2.6 Processing Queries in Embedded Languages

The embedded query languages are used for accessing the information from the databases. The host language is used for processing of this data. Host languages such as C or PL/1 are best utilized when they process data record-at-a-time. However, the advanced query languages process data a set-at-a-time. SQL takes care of the disparity by introducing the concept of a cursor. The query is processed by embedded SQL, the set (or pointers to the records of the set) is stored internally, transparently to the user. The user, through the data structure 'cursor' retrieves records from the set one-at-a-time. Essentially, the cursor is a pointer to the current record |6|.

2.7 Justification for CQL

The record oriented interactive query languages (IQLs) are more difficult to use for most applications than the set

oriented languages. This is because the queries do not contain looping constructs, and for many problems operations on the whole set are sufficient. If the IQL is not sufficiently powerful, the user has to use the embedded query language and a host language. This will make the solution inflexible, because every time a change to the query is needed, the program has to be rewritten, recompiled, relinked and tested. This is a long process. CQL solves the problem by providing the ad-hoc query facilities with the set-at-a-time retrieval mechanism, and also provides a procedural mechanism for the problems that are not solved easily by the set operations. Essentially, CQL provides both set-at-a-time and record-at-a-time functionality without forcing the user to use the looping constructs (the looping constructs are not needed because for each record CQL may invoke a CQL subroutine, the looping is implied).

The user has to learn only one language (CQL) and any changes are handled within the session. Since CQL is interactive, no recompilation and relinking is necessary.

While there are some query languages that have procedural capability, they exist to allow the user to write user friendly interfaces to the DBMS and to customize the DBMS to

specific applications by constructing queries by having dialogs with users. One of such systems is INQUIRE |9|.

However, these languages have no capability to manipulate data in records. They are, generally, just front ends to the underlying query language.

2.8 Future of Query Languages

There are several problems of interest in the area of query languages.

- Optimization of queries and updates needs to be improved, especially for use in the engineering systems where large (compared to the business databases) amounts of data need to be accessed.

- Better selection mechanisms need to be developed to search through the 'non-data processing' data types, such as graphical and textual data.

- The Natural Language front ends need to accept a larger subset of English.

The next major advance in the query languages will probably be a more user friendly interface to the CAD/CAM databases. Possibly the query languages will replace many data processing functions currently implemented by the procedural languages. When speech recognition systems become widely available and reliable, the Natural Language query languages will become much more important, particularly in the use of public information systems, such as electronic phone books, libraries, mailorder catalogs.

CHAPTER 3

Architecture

3.1 Major Components

CQL consists of five major components.

- Parser -- It is a table driven parser that receives tokens from the lexical analyzer and produces internal code for the executor under the direction of the syntax tables.

- Syntax Tables -- They define the syntax of CQL to the parser and contain the tokens used in generating the internal code.

- Lexical analyzer -- It reads the CQL source and converts the keywords into tokens.

- Executor -- It executes the code produced by the parser. It can call the parser and then call itself recursively to execute subroutines.

- Low level DBMS routines -- These routines are called by the executor to access databases. They are not a part of this project, but are the result of another thesis |12|.

3.2 Interfaces

The relationship between the parts of CQL are shown in Figure 3 on page 46. The main program calls the parser to parse the statement that the user entered at the terminal. The parser creates the internal code and returns to main. The main then calls the executor to execute the statement. For some statements it will call the DBMS routines to access the databases. It may also call the parser and itself to process the subroutines. On return from the executor, the main routine repeats the process.

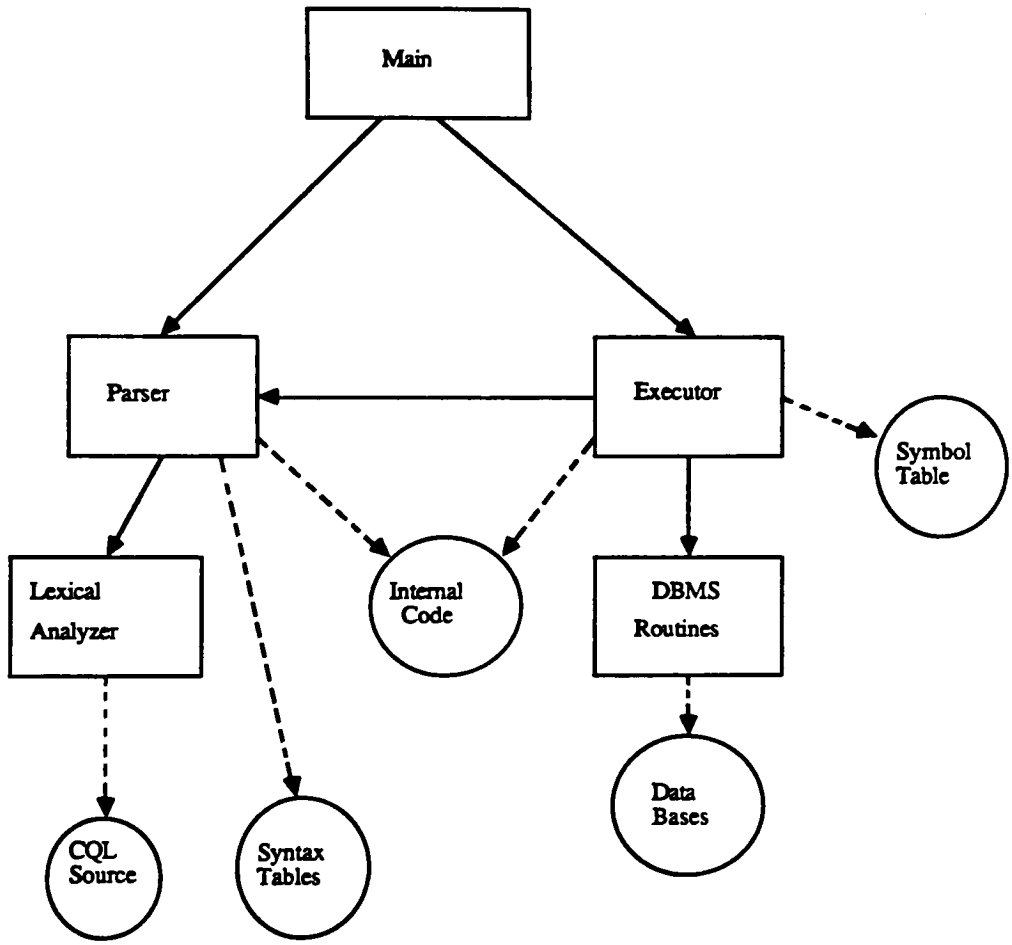


Figure 3. The relationship between major parts of CQL.

3.2.1 Parser and Executor

The code for the subroutines is stored in the internal code table. The code is identified by the subroutine's name. Once the subroutine is parsed, its code remains in the table until the user exits from CQL. Thus, even if the subroutine is executed several times, it is parsed only once.

The parser stores the internal code for the statement typed at the terminal in a special table for immediate commands.

The executor processes the code for the subroutines from the subroutine database, and for the immediate statements from the special table.

3.2.2 Main and DBMS

When CQL gets control, it creates a link to the DBMS routines (transparently to the user). When the user exits CQL, it severs this link. Two functions are provided for this purpose:

3.2.2.1 D_LOGON

```
housekp = D_LOGON(" ", " ")
```

This function establishes the link with the DBMS. The parameters must be two strings, whose content is ignored. The function returns the pointer to the user housekeeping block that is maintained by the DBMS. The only responsibility of CQL is to pass this pointer to every DBMS routine in order to identify the link (the user).

3.2.2.2 D_LOGOFF

```
D_LOGOFF(housekp)
```

This function severs the link with the DBMS. The parameter is the pointer received from the D_LOGON.

3.2.3 Executor and DBMS

The executor requests data from the DBMS routines. The following is a list of routines and a description of the calling conventions. |12| In the following descriptions, 'housekp' is the pointer to the housekeeping block which is returned by the D_LOGON.

3.2.3.1 D_OPEN

```
dbdtab = D_OPEN(dbname, pw, mode, housekp)
```

This statement creates a connection to the database whose name is passed in the string 'dbname'. It establishes the control blocks that will later be used by the DBMS to process the requests against the database. The password should be a null string. The mode is a character variable containing 'r'. The subroutine returns a pointer to the database definition table, 'dbdtab'. This pointer is used to specify the database for all other requests for the database. If the returned pointer is NULL, then there was an error.

3.2.3.2 D_CLOSE

```
error = D_CLOSE(housekp, dbdtab)
```

This closes the connection to the database specified by the 'dbdtab' and releases the control blocks allocated by the D_OPEN. If 'error' is -1 then the close was unsuccessful.

3.2.3.3 D_MARKKEY

```
error = D_MARKKEY(housekp, dbdtab, field, value, key)
```

This subroutine translates the field name and the field value into an internal representation of the key for use by the DBMS in searching of the database for the records with the key. The field name is passed by the pointer 'field' to a string containing the field name, and 'value' is a pointer to a string containing the value of the field. The field must be a keyed field. The internal key value is moved by D_MARKKEY into a string pointed to by 'key'. If there was a problem, 'error' is set to -1.

3.2.3.4 D_FNDKEY

```
error = D_FNDKEY(housekp, dbdtab, key, 1, &key_count)
```

This subroutine searches the index file of the database for the records that match the key, and passes the number of records found with the key in 'key_count'. It also stores the pointers to key records in the internal control blocks. If there was an error, 'error' is set to -1.

3.2.3.5 D_GETKEY

```
error = D_GETKEY(housekp, dbdtab, &data_offset)
```

This subroutine reads the next record in the index file to find the address of the appropriate record in the data file. The address of the data record is returned in 'data_offset' pointer. This pointer is used only by the DBMS, and CQL's responsibility is only in passing it to D_GETDAT.

3.2.3.6 D_GETDAT

```
error = D_GETDAT(housekp, dbdtab, data_offset)
```

This subroutine reads the data record from the data file and prepares the buffers for D_FLDLOC. The 'data_offset' is a pointer obtained from the D_GETDAT.

3.2.3.7 D_FLDLOC

```
D_FLDLOC(housekp, dbdtab, field_name,
         &field_table, &field_data)
```

This subroutine retrieves the value of the field from the buffer created by D_GETDAT. The field name is passed by a pointer to a character string 'field_name'. The pointer to the value is put into the 'field_data' pointer variable, and the pointer to the field table for the fields is returned in 'field_table' pointer variable. The table contains a field 'type' that contains 'C' if it is a character field, and 'I' if it is an integer field.

3.3 Internal Data Structures

3.3.1 Syntax Tables

These tables define the syntax of the CQL for the parser. There is one table for each statement of the CQL. The tables are character strings. Each character position contains the directive for the parser.

The specific contents of these tables are discussed in the next chapter, Design of CQL.

3.3.2 Token Table

CQL has two types of keywords: simple and complex. Each simple keyword means a specific thing, depending on its position within the sentence. For example, 'and' keyword is an operator in the expression, and '+' is a keyword that specifies a binary plus operation.

The complex keywords precede the constructs of CQL, such as 'while' precedes the rest of the 'while' statement, and 'print' precedes the 'print' statement or the 'print' option of the 'select' statement.

The token table defines:

- keyword
- its token
- pointer to the appropriate syntax table if it is a complex keyword, or a NULL pointer for the simple keywords.
- rank, input and output precedences -- these are used in parsing expressions and will be explained in the description of the expression parser.

3.3.3 Action Table

For each CQL statement, the executor contains a subroutine that executes the statement. This table is a cross reference of the statement tokens (the ones that identify the state-

ment) and of the address of the executor's subroutine for the statement.

3.3.4 Variable Symbol Table

This is the symbol table used by the executor that contains the values of the local variables. It either contains a pointer to a character string if the variable holds the character value, or a number if the variable has a numeric value.

3.3.5 Subroutine Symbol Table

This table contains the address of the internal code of the translated CQL subroutine. Once the subroutine is parsed, its name and address are entered into the table. They are retained until the end of the CQL session.

3.3.6 Database Symbol Table

This table contains the pointer to the database housekeeping block returned by the D_OPEN. Once the database is opened, its name and a pointer to the housekeeping block are entered

into the table. They stay in the table until the database is closed.

CHAPTER 4

Design of CQL

In the discussion of the design of the CQL it would be useful to refer to the data flow diagram of the CQL in Figure 4 on page 58 and Figure 5 on page 59.

4.1 Lexical Analyzer

The lexical analyzer reads the next item from the input stream, determines what kind of item it is, and returns its findings in the data structure CUR_TKN. It recognizes four types of items:

1. Alphabetic keyword -- a string beginning with an alphabetic character and followed by 0 or more alphanumeric characters (up to eight total). It optionally can be followed by a period and another string. The lexical analyzer stores the name in the CUR_TKN structure.

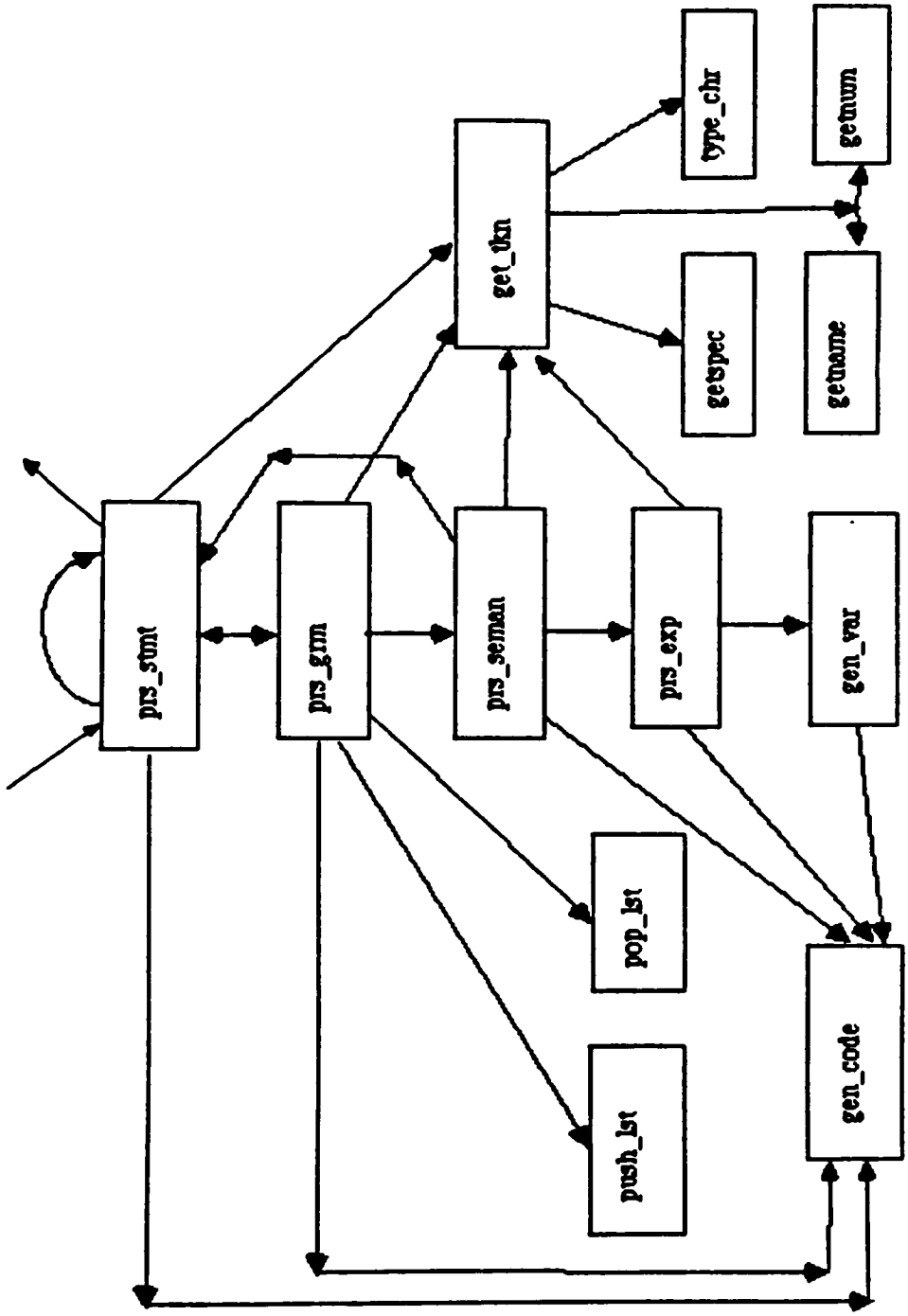


Figure 4. Data flow diagram of CQL parser

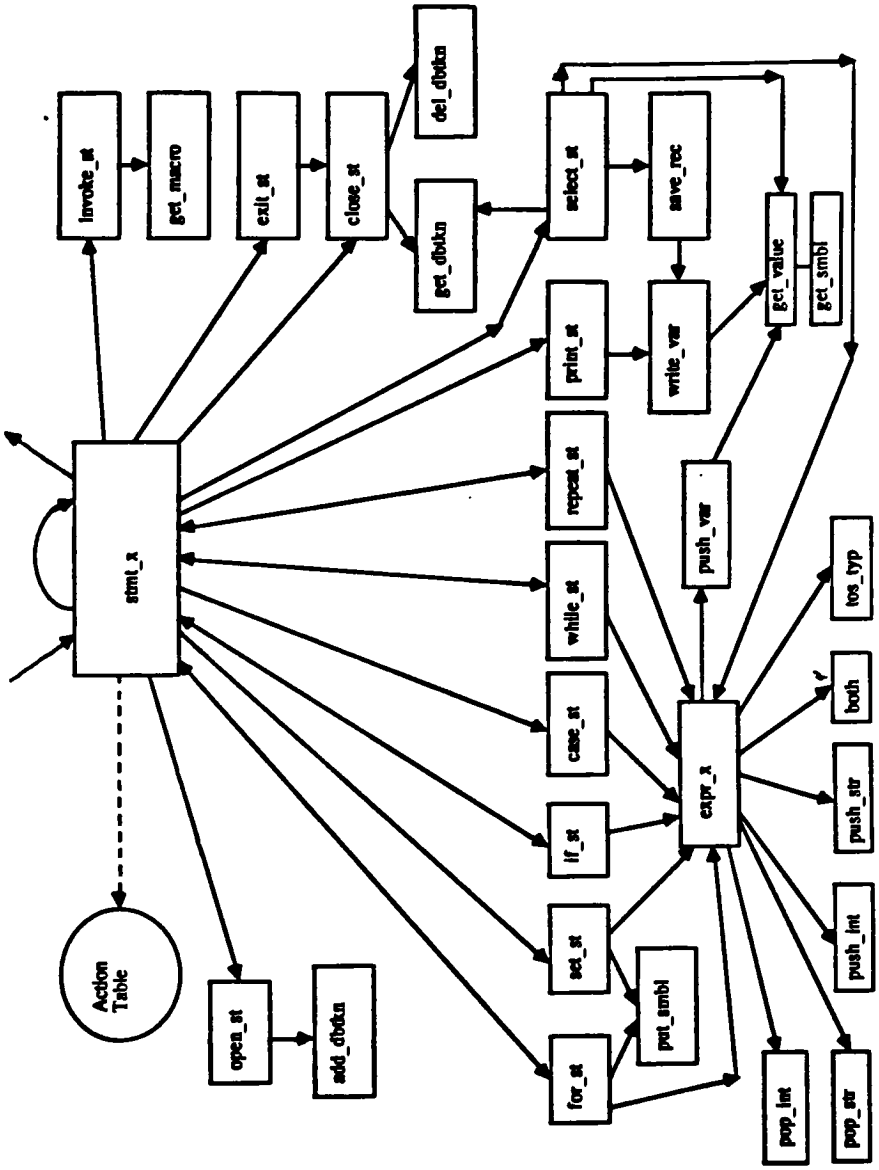


Figure 5. Data flow diagram of CQL executor

2. Number -- an unsigned decimal integer. Must be terminated by a non-alphanumeric character. The lexical analyzer stores the value in the CUR_TKN and sets the token to integer.

3. Special character keyword -- can be one or two characters long. The lexical analyzer looks up the token in the token table. If the first character of the keyword is followed by another special character, and if these two characters form a known keyword, this keyword is used. Otherwise the first character determines the keyword.

   For example, '>=' forms a token 'greater or equal', whereas '*(' forms two tokens 'multiply' and 'open parenthesis'.

4. Character string -- a string surrounded by the double quotes and followed by a non-alphanumeric character. The character string is added to the string database, and its address is put into the CUR_TKN.

Sometimes it is necessary for the parser to look ahead by one token. If this token is not used by the parser at this

point, the parser can set a flag to tell the lexical analyzer not to do anything next time it is called, but to return the token that is already in CUR_TKN.

An example is:

```
PRINT A B C;
```

The parser gets 'PRINT' token, then 'A', 'B', 'C' and ';' tokens. When it sees the ';', the parser knows that it is the end of the variable list and marks the current token ';' as not used. The next time the lexical analyzer is called, it will return the token ';'.

4.2 Parser

4.2.1 Syntax Tables

Syntax tables are the templates over which the input statements are matched. There is one syntax table for each CQL statement. The tables are character strings, every character of which tells the parser what type of CQL construct has to be matched. There are four types of directives in the syntax tables.

1. terminal -- when the parser encounters this type of directive, it expects a matching token in the input source statement. If it does not match, then there is a syntax error.

2. non-terminal -- when the parser encounters the non-terminal token, it means that the input statement must have a construct that matches the non-terminal. There are three types of non-terminal directives:

   a. non-terminals that provide a syntax table -- they cause the parser to call itself recursively to process the portion of the source statement under the direction of the syntax table. One may think of this as the 'call' to another syntax table.

   b. statement non-terminal -- this directive does not provide a syntax table, it just means that any CQL statement can match it. The parser recursively calls the statement processor to identify the statement in the source and then parse it.

c. expression non-terminal -- this directive is matched by the infix expressions. The parser calls the expression translator to process expressions.

3. option -- this directive has a matching 'option end' directive further in the syntax table. These directives enclose an optional part of the statement. It provides a simple backtracking method. Once this directive is encountered, if the next directive in the symbol table matches, the parser is committed to using this portion of the syntax table. If the first directive after the 'option' is not matched, the parser skips over the optional part until it finds the matching 'option end' directive.

4. list directives -- they tell the parser that the enclosed definition can be repeated 1 or more times. There are two types of lists. A comma list is a list of the like objects separated by commas. A blank list is a list of like objects separated by blanks. In the syntax table, the 'end list' directives identify where the end of the list is. The 'end comma list' directive tells the parser that a comma is expected, otherwise it is the end of the list. For the blank list, the 'end blank list'

directive tells the parser that a list is terminated when a special character token is encountered. For either list, if the parser expects the next element in the list, it restarts processing of the syntax table at the top of the list. If it processed the last element of the list, the parser continues with the directive following the 'end list' directive.

For example, syntax table for the SELECT statement is (each line is a character in the table):

```
EXPR_TKN            non-terminal
COMMA_TKN           terminal
LIST_TKN            list directive
   OPT_BEG_TKN      option directive
      PRINT_TKN     non-terminal
   OPT_END_TKN      option directive
   OPT_BEG_TKN      option directive
      INVOKE_TKN    non-terminal
   OPT_END_TKN      option directive
   OPT_BEG_TKN      option directive
      ATEND_TKN     non-terminal
   OPT_END_TKN      option directive
   OPT_BEG_TKN      option directive
      SAVE_TKN      terminal
      ID_TKN        terminal
   OPT_END_TKN      option directive
ECLST_TKN           list directive
FIN_TKN
```

4.2.2 Format of the internal code

The parser converts the source into the internal code. The internal code is a string consisting of tokens that direct the executor in its actions. The representation matches the

source statement, except for the strict format. The first token is the statement token, i.e. it identifies the statement that follows it. After it comes the length of the translated statement. It is used in order to quickly find the end of the statement. After the length comes the rest of the statement. The important parts of the statement are simple and database variables, integers, expressions and statements.

The simple and database variables consist of the variable or DB variable token, followed by the 8 or 16 character string containing the variable name or the database name followed by the field name. The names are padded with NULLs on the right to fill in all 8 bytes.

The integers are preceded by the integer token, followed by the four byte integer. The number is stored as a four byte character and should be moved to the integer variable before calculations are done on it. This may be necessary on the machines where the operand alignment is important. The CQL executor moves them to the integer variables, so that the code is machine independent.

The expressions are represented by the expression token, followed by the length of the expression, followed by the expression in the reversed polish notation. Expressions consist of operator, variable and integer tokens.

Statements are in the format that is discussed in this section. The block statement consists of a group of statements enclosed in '{' and '}' tokens. The program ends with the statement token FIN_TKN. Here is an example of the statement:

```
while x < 21
   { print x;
     set x = x + 1;
   };
```

in the internal representation:

+ 114 +	while token
+ 69 +	2 bytes stmt length = 69
+ 51 +	expression token
+ 17 +	2 bytes expression length
+ 52 +	variable token
+ 'x' +	8 bytes of padded var name
+ 57 +	integer token

21	4 bytes of integer
31	'<' (less than) token
19	'{' (start block stmt) token
47	length of statement
109	print token
10	length of statement
52,'x'	var token, variable name
116	set statement token
29	length of statement
52,'x'	destination variable
21	'=' (equal) token
51	expression token
17	expression length
52,'x'	variable 'x'
57,1	integer '1'
22	'+' (plus) token
20	'}' (end block stmt) token
0	end of program token

4.2.3 Statement Processor

When the parser is entered, it expects to receive a statement from the input stream. The first token determines the

statement. The statement processor searches the token table for the entry that matches the token. The entry has a pointer to the appropriate syntax table. The statement processor then calls the grammar processor that will parse the rest of the statement under the control of the syntax table. Then it checks if the statement is terminated by the ';' token. This part of the parser determines if it is a block statement (if it starts with '{'). If it is, it calls itself repeatedly to process each statement inside a '{}' pair. Otherwise it calls the grammar processor. It also calculates the length of the statement, and stores it at the start of the translated statement, where it previously reserved two bytes for the length.

4.2.4 Grammar Processor

The grammar processor gets the address of the syntax table appropriate for the statement. It then requests tokens from the lexical analyzer.

The function of the grammar processor was discussed in some detail in the section 'Syntax Tables'. Let's elaborate further on how it functions.

If the current entity in the statement is an alphabetic keyword, then there can be some confusion on whether it is a variable name or a keyword. For example, 'and' can be a variable or an 'and' operator depending on where it appears in a statement. The grammar processor treats it as a variable if the syntax tables direct it to match a variable, otherwise the search is made in the token table for the matching token.

This processor handles the special commands, list and option directives discussed in the 'Syntax Tables' section. Every time the processor sees the list directive, it stacks the directive. When the list is processed, the entry is popped off the stack. This method allows nested lists and makes the syntax tables more compact.

For the non-terminal directives, it determines the kind of non-terminal it is.

o statement -- the grammar processor calls the statement processor, which actually called the grammar processor

- expression -- the grammar processor calls the expression processor to convert infix expressions into reversed polish expressions.

- the ones that provide a syntax table -- call the statement processor with a flag not to search for the closing ';'. The statement processor is called so that the entity's length is recorded in the internal code.

4.2.5 Expression Translator

The expression translator converts the infix expressions into the postfix equivalent. It scans expressions from left to right and uses stack to change the order in which tokens are generated. The end of expression is determined when a token is encountered that would make the expression invalid. In order to determine the order in which to generate the code, each token in the expression is assigned a set of numbers:

- Input Precedence. This is the precedence of the token that is used by the translator when the token comes off the input expression. It is compared with the stack precedence of the token on top of stack. The higher the

precedence, the earlier token will be generated into code.

- Stack Precedence. This is the precedence of the token once it has been pushed onto stack.

- Rank. It is used to determine if the expression is valid. Variables and constants have rank of 1. The operators are assigned the negated (number of operands required by the operator minus 1). For example, binary '*' (multiply) is assigned the rank of -1, whereas a unary 'not' is assigned rank of 0. Every time a token is generated into code, the rank of the token is added to the total rank. If the total rank ever falls below 1 there is a problem with the expression. After the expression is fully processed, the total rank must equal 1.

The algorithm of the expression translator |10|:

1. Push '(' on stack.

2. Repeat steps 3 through 5 until end of expression

3. Get next token. If no tokens left in expression, make the token ')'

4. While input precedence is less than the stack precedence of the top of stack token, pop token from stack and generate code and add its rank to total rank. If total rank is less than 1, then issue error message and exit.

5. If input precedence equals stack precedence, pop stack (it means that top of stack holds '(' while the input token is ')').
   else push the input token onto the stack.

6. Total rank must be 1, else report error.

The '(' and ')' are not generated, since they are not needed in the postfix expressions. The input precedence of the '(' is set very high, so that nothing is popped off before the '(' is pushed onto the stack. Its stack precedence is set very low, so that it is not popped off until a matching ')' is encountered on input.

The ')' has a very low input precedence (equal to the stack precedence of '(' so that the parser can easily determine

when a matching ')' is read). This causes all tokens that are above the '(' on the stack to be popped off and generated. Then, when the top of stack holds the ')', the top of stack is discarded, and the next input token is read.

The input precedences of variables, constants and left associative operators is lower by one than their stack precedences. This forces tokens with the same precedence to be generated in the left to right order. The right associative operators have the input precedences be higher by one than the corresponding stack precedences, thus causing the order of generation to be right to left.

4.3 Executor

The executor executes the internal code. For each statement, the executor contains the subroutine that processes it. The first token in the statement defines the statement. The executor looks up the subroutine address in the action table -- the table that matches the statement tokens with the corresponding subroutines.

Each subroutine is called with the address of the first statement token. The subroutines return the address of the

last character position of the statement after executing it. The subroutines are fairly straightforward. They perform the functions described in Chapter 1, where the CQL statements were defined.

4.3.1 Expression Evaluator

The postfix expressions are evaluated by the following algorithm:

```
if the current token is an operand, stack it
else (it must be an operator)
   perform the operation on as many stack elements
   as the operator requires.
   stack the result
repeat the above until end of expression
if one element on the top of stack,
then it is the result.
otherwise there was an error
```

The variables can be either character or integer variables. The type is determined by the value that is assigned to them. A variable can change its type from one to another

when the assignments are made. For example, if variable 'x' contains a string "abcd", and we execute the statement

```
set x = x < "wxyz";
```

then the value of 'x' will become an integer with the value 1 (TRUE). Operations on the mixed types is not allowed. For example:

```
set x = 1 < "1";
```

is not allowed, while

```
set x = (1 = 1) or ("1" > "2");
```

is allowed, because each comparison creates an integer result.

CHAPTER 5

Conclusions

5.1 Trade-offs -- Intermediate Code

There are two types of intermediate code that could be used for CQL.

The first type is code that corresponds closely to the source. This was used in CQL. Each statement is translated into a tokenized form that has a strict format that is easy for the executor to interpret.

The second is pseudo machine that consists of a set of simple operations that can form a program that can accomplish anything that CQL can. CQL source would be translated into a program in this P-code. The executor would have to interpret P-code.

The advantages of the first type are:

1. The interpreted code is easy to convert back into the source, thus making list, edit and trace functions easy to implement without having to go back to the source files. These are desirable features for the interactive language. The programs can be saved in the tokenized form and then converted back to CQL source if necessary.

2. It occupies less room than the P-code version. This is because the tokens are more specific, thus fewer are needed to define operations.

3. Simpler parser. This is because the translated code corresponds closely to the source. For P-code the syntax tables would have to be more complex by including the P-code that would have to be generated and adding the code to the parser that could interpret these more sophisticated tables.

4. Faster parser. Because the P-code parser would have to do the work of this parser plus generate several P-code instructions for each statement.

5. Faster executor. Since each statement has a corresponding subroutine in the executor, it can be optimized to execute each statement as fast as possible. With P-code the executor would have to interpret more instructions, thus making the overhead higher.

6. Easier to make changes to the parser, since the syntax tables are simpler.

The disadvantages of the first method are:

1. More complex executor. The P-code executor only has to interpret a set of simple P-code instructions, whereas the this executor has to have a subroutine for each statement.

2. More difficult to make changes to the executor, since most changes in CQL require corresponding change in the executor. The P-code executor would not have to be changed under most circumstances.

3. The P-code generator could become, with few modifications, a real code generator, thus turning CQL inter-

preter into a compiler. It would be more difficult to convert this interpreter to generate real code.

Since CQL is an interactive language, the advantages outweigh the disadvantages. The advantages 1, 4 and 5 are the most important for the interactive environment.

5.2 Future Enhancements

There are several enhancements that can be added to the CQL. Some are simple and some could constitute enough work for another thesis.

5.2.1 Omit Redundant Tokens

Some generated tokens are not necessary for the executor. For example, the 'set' statement generates code

```
<set> <variable> <=> <expression> ...
```

The '=' token is not necessary. The parser generates extra tokens for easier debugging of the parser. Of course, the executor has to skip over the redundant tokens.

5.2.2 Optimize Scanning of the Syntax Tables

Current implementation of the parser skips to the end of the options lists in the syntax tables if the option was not chosen. The skipping is done by looking at each directive until the 'option end' directive is reached. This operation can be sped up by including the length of the option in the syntax table, thus making the skip a very quick operation.

5.2.3 More Efficient Table Searching

CQL searches its internal tables sequentially. To speed up parsing and execution stages a more efficient searching algorithm could be implemented. Such as hashing or binary trees.

5.2.4 Stop Forcing ';' to end every statement

CQL forces ';' to terminate every statement. Actually the language would not be ambiguous if that requirement were waived under certain conditions, such as the last statement of the { ... } group, and after this group.

5.2.5 Add more commands to CQL

Add some commands that would make debugging and developing of CQL programs simpler:

- LIST -- this command would decompile the tokenized statements and display them on the terminal.

- DELETE -- this command would remove a macro from the internal code table database. It would be useful if the source file were modified and user wanted to execute the new version.

- EDIT -- this command would decompile the tokenized statements end call some editor to edit it.

- TRACE -- this command would trace the execution of the CQL statements, performing the List command on the executed statements and optionally display values of the changed variables.

- BREAK -- this would suspend execution at the statement and allow user to 'look around'. The user would tell CQL to stop at a certain statement, or when certain condi-

tions occur, such as some variable is changed or attains a certain value, or some statement is executed a certain number of times.

- GET -- permit input of data from flat files or terminal.

5.2.6 Add more string manipulation functions.

At this point, CQL allows only assignments and comparisons of strings. Some useful functions include

- concatenation
- substring
- difference
- conversion to numeric and back
- translation from one character set to another, like uppercase from mixed.

5.2.7 Add more operators

- Useful operators would be set comparisons (does variable match one of the list of variables).

- Exponentiation

- Remainder

5.2.8 Use strings instead of identifiers

In some statements, use strings instead of identifiers, so that users can calculate the values dynamically. Examples are:

- OPEN and CLOSE statements -- database name and password

- INVOKE statement and INVOKE and ATEND options of the SELECT statement -- the macro name

5.2.9 Make PRINT statement more powerful

Instead of PRINT statement printing on each line the variable name followed by the value, allow formatting of the output and permit expressions in the statement. Also add an option to let output go to a file.

5.2.10 Make SELECT statement more powerful.

- Allow an unlimited expression as a selection criterion instead of limiting it to one database.

- Add report generator to allow sorting and formatting output of the queries.

- Allow searching of the flat files.

- Allow creation of the temporary views.

APPENDIX A. BNF NOTATION FOR CQL

```
<alpha>            ::= a|b|c|d|e|f|g|h|i|j|k|l|m|
                       n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>            ::= 0|1|2|3|4|5|6|7|8|9
<alphanumeric>     ::= <alpha>|<digit>
<any char>         ::= any ASCII character
<integer>          ::= <digit>|<digit> <integer>
<unquoted string>  ::= <any char>|
                       <any char> <unquoted string>
<string>           ::= "<unquoted string>"
<constant>         ::= <integer>|<string>
<identifier>       ::= <alpha>|<identifier><alphanumeric>
<local variable>   ::= <identifier>
<database name>    ::= <identifier>
<database field>   ::= <database name>.<identifier>
<variable>         ::= <local variable>|<database field>
<expr>             ::= <gen expr><log_op><gen expr>|
                       <gen expr>
<gen expr>         ::= <gen expr><rel_op><arith expr>|
                       <arith expr>
<arith expr>       ::= <arith expr><add_op><term>|<term>
<term>             ::= <term><mult_op><factor>|<factor>
<factor>           ::= <variable>|<constant>|<expr>|
```

```
                    !<factor>
<log_op>          ::= and|or
<rel_op>          ::= =|!=|<|>|<=|>=
<add_op>          ::= +|-
<mult_op>         ::= *|/
<expr list>       ::= <expr>|<expr list>,<expr>
<stmt list>       ::= <stmt>|<stmt list> <stmt>
<stmt>            ::= <block statement> |
                      <case stmt> |
                      <close stmt> |
                      <exit stmt> |
                      <for stmt> |
                      <if stmt> |
                      <invoke stmt> |
                      <open stmt> |
                      <print stmt> |
                      <repeat stmt> |
                      <select stmt> |
                      <set stmt> |
                      <SYS stmt> |
                      <while stmt>
<block stmt>      ::= { <stmt list> };
<case stmt>       ::= case { <case list> };|
```

```
                        case { <case list> other:<stmt>};
<case elem>         ::= (<expr list>):<stmt>
<case list>         ::= <case elem>|<case list><case elem>
<close stmt>        ::= close <db list>;
<db list>           ::= <database name>|
                        <db list>,<database name>
<exit stmt>         ::= exit;
<for stmt simple>   ::= for <variable> = <expr> to <expr>
<for stmt ctl>      ::= <for stmt simple> |
                        <for stmt simple> step <expr>
<for stmt>          ::= <for stmt ctl> <stmt>;
<if stmt then>      ::= if <expression> <stmt>
<if stmt>           ::= <if stmt then>;|
                        <if stmt then> else <stmt>;
<invoke stmt>       ::= invoke <identifier>;
<open stmt>         ::= open <db list>;
<print stmt>        ::= print <var list>;
<var list>          ::= <variable>|<var list> <variable>
<repeat stmt>       ::= repeat <stmt> until <expr>;
<select stmt>       ::= select <special expr>
                          <select options>;
<special expr>      ::= <db variable>=<string> |
                        <db variable>=<string>and<expr>
<select print>      ::= ,print <var list>
```

```
<select invoke>   ::= ,invoke <identifier>
<select atend>    ::= ,atend <identifier>
<select save>     ::= ,save <identifier>
<select option>   ::= <select print>|
                      <select invoke> |
                      <select atend> |
                      <select save> |
<select options>  ::= <select option>|
                      <select options><select option>
<set stmt>        ::= set <variable> = <expr>;
<sys stmt>        ::= sys <unquoted string>;
<while stmt>      ::= while <expr> <stmt>;
```

References

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM Vol. 13, No. 6., pp. 377-387.

2. Zloof, M. (1977). "Query-by-Example: A Data Base Language". IBM Systems Journal, Vol. 16, No. 4.

3. Date, C. J. (1982). "An Introduction to Database Systems". Third Edition, publisher Addison-Wesley.

4. Reisner, P. (11/1984). "Measurement of SQL: Problems and Progress". Unpublished Technical Reports.

5. Held, G. D., Stonebraker, M. R. and Wong, E. "INGRES - A Relational Data Base System". in "Data Base Management Systems", edited by Schneiderman, B. AFIPS Press.

6. IBM manual (12/1984). SQL/DS Application Programming for VM/System Product, SH24-5068-0.

7. Tsichritzis, D. C. and Lochovsky, F. F. (1977) "Data Base Management Systems". Academic Press.

8. Ullman, J. D. (1982) "Principles of Data Base systems." Second Edition. Computer Science Press.

9. Infodata manual (11/1981). "User Language Tutorial"

10. Tremblay, J. P. and Sorenson, P. G. (1976). "An Introduction to Data Structures with Applications". McGraw Hill Computer Science Series. "Problems and Some Solutions in Customization Database Front Ends". ACM Transactions on Office Information Systems. Vol 3, No 2.

11. Deutsch, Donald (01/07/86). Private communication.

12. Fabbio, Robert. (1984) "Relational-Like File Structure" Masters thesis. RIT

13. Jarke, M. and Vassiliou, Y. (9/1985). "A Framework for Choosing a Database Query Language". ACM Computing Surveys, Vol. 17, No. 3., pp. 313-340.